2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing

Modelling Performance & Resource Management in Kubernetes

Víctor Medel
Aragon Institute of Engineering Research (I3A)
University of Zaragoza, Spain
vmedel@unizar.es

Omer Rana
School of Computer Science & Informatics
Cardiff University, UK
ranaof@cardiff.ac.ukm

José Ángel Bañares
Aragon Institute of Engineering Research (I3A)
University of Zaragoza, Spain
banares@unizar.es

Unai Arronategui
Aragon Institute of Engineering Research (I3A)
University of Zaragoza, Spain
unai@unizar.es

ABSTRACT

Containers are rapidly replacing Virtual Machines (VMs) as the compute instance of choice in cloud-based deployments. The significantly lower overhead of deploying containers (compared to VMs) has often been cited as one reason for this. We analyse performance of the Kubernetes system and develop a Reference net-based model of resource management within this system. Our model is characterised using real data from a Kubernetes deployment, and can be used as a basis to design scalable applications that make use of Kubernetes.

1. INTRODUCTION

Kubernetes [1] provides the means to support container-based deployment within Platform-as-a-Service (PaaS) clouds, focusing specifically on cluster-based systems. Kubernetes enables deployment of multiple "pods" across physical machines, enabling scale out of an application with dynamically changing workload. Each pod can support multiple Docker containers, which are able to make use of services (e.g. file system and I/O) associated with a pod. With significant interest in supporting *cloud native applications* (CNA), Kubernetes provides a useful approach to achieve this. One of the key requirements for CNA is support for scalability and resilience of the deployed application, making more effective use of on-demand provisioning and elasticity of cloud platforms. Containers provide the most appropriate mechanism for CNA, enabling rapid spawning and termination compared to Virtual Machines (VMs). The process management origin of container-based systems also aligns more closely with the granularity of many CNA – enabling single or groups of containers to be deployed on-demand [1].

Stream processing represents an emerging class of applications that require access to Cloud services where deployment overhead of launching/ deploying new VMs or Containers remains a significant challenge. Amazon Lambda [2] provides an example of such a system, where resource provisioning is carried out at 100ms intervals (rather than on an hourly interval as with most Cloud PaaS and IaaS providers). In AWS Lambda, events generated through one or more user streams (via other AWS services, e.g. S3, DynamoDB, Cognito Authentication for mobile services etc or via a user developed application) are processed through a *lambda* function. This function is provisioned through a container-based deployment, where the execution of the lambda function is billed at 100ms intervals. The container can be *frozen* when no longer in use (whilst maintaining a temporary storage space and link to any running processing). Understanding performance associated with deploying, terminating and maintaining a container that hosts such a lambda function is therefore significant, as it affects the ability of a provider to offer more finer grained charging options for users with stream analytics/ processing requirements.

We present a Reference Net (a kind of Petri Net [2] representation) based performance and management model for Kubernetes, identifying different operational states that may be associated with a "pod" and container in this system. These states can be further annotated and configured with monitoring data acquired from a Kubernetes deployment. The model can be used by an application developer/ designer to: (i) evaluate how pods and containers could impact their application performance; (ii) used to support capacity planning for application scale-up.

2. RELATED & BACKGROUND WORK

Virtual Machine virtualization and container virtualization have attracted considerable research attention focusing on performance comparison using a suite of workloads that stress CPU, memory, storage and networking resources [3, 4, 5]. However, to the authors' knowledge any work has addressed container performance issues following a rigorous

[1] http://kubernetes.io/

[2] https://aws.amazon.com/es/lambda/details/

UCC '16, December 06 - 09, 2016, Shanghai, China

ACM ISBN 978-1-4503-4616-0/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2996890.3007869

analytical approach. Unfortunately, most computer scientist are either not familiar with or reluctant to use formal methods. Even mature technologies, such as cloud computing, provides a small portion of the work done on performance considering formal models [6]. In [7, 8] is proposed an iterative and cyclic approach, starting from the specification of functional algorithms (specified in the functional models). Then, it continues with the specification of the computational resources available (in the operational models), providing complementary views: Control flow, dataflow, and resources. The central role in this methodology is given to a set of Petri Net (PN) models describing the required functionality and the computational resources involved in the execution. PN is a well known formalism that combines simulation and analysis techniques. The formalism allow a developer to analyse the behaviour of the system throughout the development lifecycle and to gain understanding of infrastructure and application behaviour. In particular, PNs provide different analysis and prediction techniques that allow developers to assess functional and non-functional properties by means of simulation, and qualitative/quantitative analysis. Timed Petri net enriches the model with time, which enables the exploration of minimal and maximal boundaries of performance and workload. As simulation tool, Petri nets allow the formulation of models with realistic features (as the competition for resources) absent in other paradigms (as nude queueing networks).

Anyway, these models must be feed with temporal information, which have been the focus of previous works. Performance evaluation has been done mainly for traditional virtual machine execution in Clouds [9]. In [4, 10], Virtual Machines and containers are compared attending several performance metrics. In the last few years, a few proposals have emerged to manage container clusters like Kubernetes and Docker Swarm [3]. Currently, Kubernetes seems to be the most featured and production grade. Limited research exists about container architecture & Kubernetes performance. In [5], a container performance study with Docker shows network performance degradation in some configurations and a negligible CPU performance impact in all configurations. Network virtualization technologies (Linux Bridge, OpenvSwitch) are pointed to as reasons, but mainly in full nested-container configurations where network virtualization is used twice. Unlike Docker, Kubernetes uses a partial nested-container approach with the Pod concept where network virtualization is used once, as the same IP address is used for all containers inside a Pod, leading to better performance. The Kubernetes team reports several performance metrics[4]. They measured the response time of different API operations (e.g GET, PUT, POST operations over nodes and pods) and the Pod startup end-to-end response time. In their experiments, the 99th percentile pod startup time was below three seconds in a cluster with 1000 nodes. Also, they propose Kubemark, a system to evaluate the performance of a Kubernetes cluster [5].

Kubernetes is based on a master-slave architecture, with a particular emphasis on supporting a cluster of machines. The communication between Kubernetes master & slaves (called *minions* in Kubernetes terminology) is realised through the *kubelet* service. This service must be executed on each machine in the Kubernetes cluster. The node which acts as master can also carry out the role of a slave during execution. As Kubernetes works with Docker containers, the *docker daemon* should be running on every machine in the cluster. In addition, Kubernetes makes use of the *etcd* project to have a distributed storage system over all nodes, in order to share configuration data. A master node runs an API server, implemented with a RESTful interface, which gives an entry point to the cluster. Kubernetes uses its API service as a proxy to expose the services executing inside the cluster to external applications/ users.

2.1 Kubernetes Background

The basic working unit in Kubernetes is a *pod* – an abstraction of a set of containers tightly coupled with some shared resources (the network interface and the storage system). With this abstraction, Kubernetes adds persistence to the deployment of single containers. It is important to note two aspect of a pod: (i) a pod is scheduled to execute on one machine, with all containers inside the pod being deployed on the same machine; (ii) a pod has a local IP address inside the cluster network, and all containers inside the pod share the same port space. The main implication of this is that two services which listen on the same port by default cannot be deployed inside a pod. A pod can be replicated along several machine for scalability and fault tolerance purposes. When a service or a set of services are deployed over several machines, we can consider: (1) *functional level* or application level involves exposing dependencies between the deployed services. Different services need to be coordinated in order to provide a high level functionality. An example of this kind of relationship is the deployment of a stream processing infrastructure (e.g. Apache Kafka, Storm, Zookeeper and HDFS for persistence) or the GuestBook example provided by Kubernetes, composed of a php frontend, a redis master and slave. Ubuntu Juju is a reference project which works at the functional level to coordinate the deployment of services. (2) *operational level* or deployment level involves mapping services to physical machines, VMs, pods or containers. It is platform dependant and must involve isolation between resources. Kubernetes primarily focuses on the operational/ deployment level. A pod implements a service, and coordination between different pods is achieved through global variables. Services running in others pods can be discovered through a DNS. This approach imposes some restrictions to Kubernetes. For instance, in the Guestbook example, Kubernetes' scheduler cannot ensure that the three pods are deployed rightly, because Kubernetes does not manage the application level. The communication between pods is made at application level. Kubernetes makes use of a number of support services, deployed in an isolated namespace `kube-system`, such as a logging (fluentd) & monitoring service (heapster & Prometheus), a dashboard (Graphana), etc. Kubernetes also has a specific DNS server, deployed as an add-on inside a pod.

3. STATE & PERFORMANCE MODELS

Kubernetes allows us to deploy several pods over physical machines while it manages their lifecycle (i.e. start/stop new instances of running pods). We model a pod lifecycle

[3]https://www.docker.com/products/docker-swarm/

[4]http://blog.kubernetes.io/2016/03/1000-nodes-and-beyond-updates-to-Kubernetes-performance-and-scalability-in-12.html

[5]https://github.com/kubernetes/kubernetes/blob/release-1.3/docs/proposals/kubemark.md

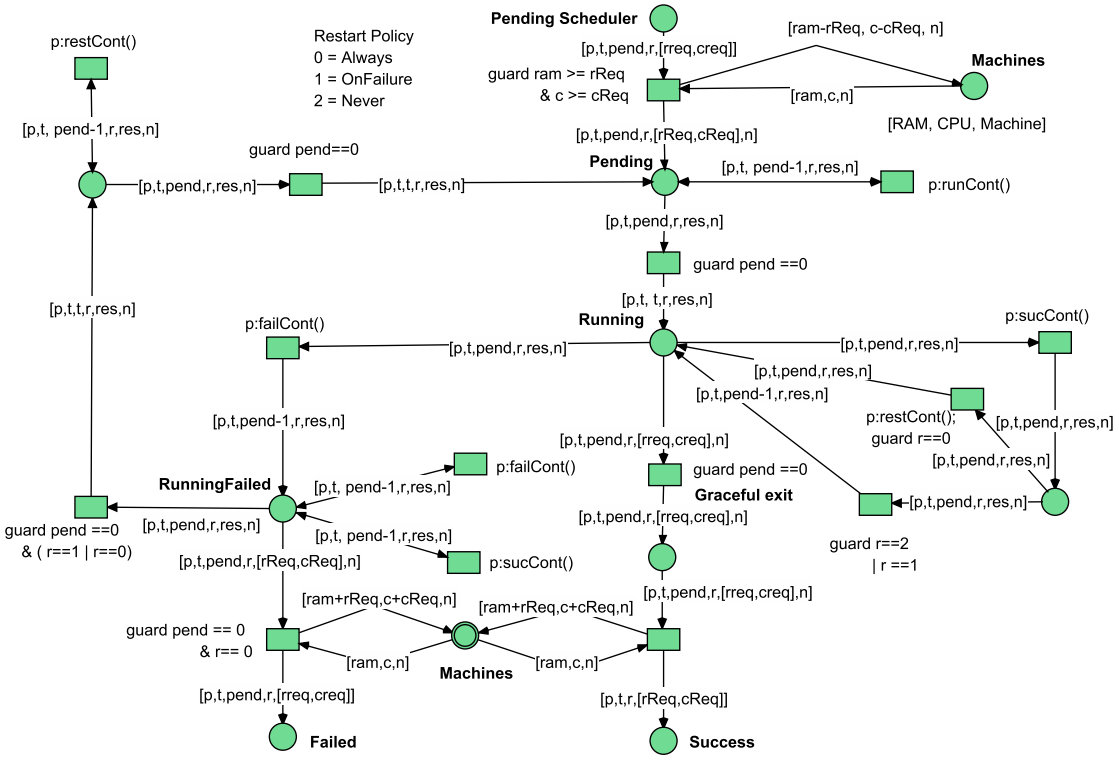

Figure 1: Model of Kubernetes manager system and shared resources

in order to estimate the impact of different scenarios on the deployment time and the performance of the applications running inside the pod. According to the pod restart policy, there are two kinds of pods in Kubernetes: (i) *Service Pods* – expected to be in a permanent running state – which will generate a background workload in the cluster, and they may include Kubernetes system services (e.g. monitoring and logging tools) or application services; (ii) *Job/batch Pods* – expected to terminate on completion. The restart policy of a job can be *onFailure* or *never*. the pod's execution time is dependant on the application deployed inside the containers. If the restart policy is *onFailure*, the time to deploy new pod instances in failure scenarios will have impact on the total performance of the job or service. The performance metrics for both kinds of pods are different. For example, for a service pod the deployment time is significant, along with the response time; while for a job pod the deployment time and the total execution time (including restarting, if necessary) are both useful metrics. Before a pod is executed, it requests resources from the Kubernetes scheduler such as RAM and CPU. If there are enough resources available in the cluster, the scheduler chooses the best node to deploy the pod. The CPU request is only taken into account when CPU-intensive processes are running. When a container is idle (e.g. it is inside a service pod and the service has low use at some point), other containers can utilise the unused CPU time. With this resource model, it is easy to see that the total performance of the pod depends on its resource requests and on the total workload at the node.

3.1 Characterising Performance Models

Petri Nets [2] are a formal modelling tool that can be used to describe and understand the behaviour of concurrent and distributed systems. A PN is a graph with two disjoint kinds of nodes: places and transitions. In high-level PNs, places represent a state of the system (or subsystem) and transitions represent the actions (or events) that change the state. Arcs going from a place to a transition represent preconditions that need to be fulfilled in order to fire the transition. Likewise, arcs from transitions to places represent postconditions fulfilled when a transition is fired.

We use Object Nets [11] (a type of PN [2]) with reference semantics for our model, where a token net represents a task or agent, with a plan for the execution procedure, that can be executed in different machines or locations. Token nets are called object nets in distinction to the system net to which it belongs. Interactions between objects nets, or between the object nets and the system nets are represented by labels as will be illustrated in the models. We have implemented our model in Renew [12] which uses Java as a language for the inscriptions and which allows us to use tuples as tokens. Additionally, in Renew, it is possible to pass parameters to synchronisation inscriptions. The inscriptions in the arcs match the tokens in the net using tuple notation. For legibility concerns in the Nets figures, a double circle place represent the place with the same name.

In Kubernetes, the lifecycle of a pod depends on the state (and consequently the lifecycle) of the containers that are inside it. For instance, a pod has to wait until all its containers have been created. With the Object Nets abstraction, we can represent the Kubernetes manager system as the System Net and the Pods (with the containers) as To-

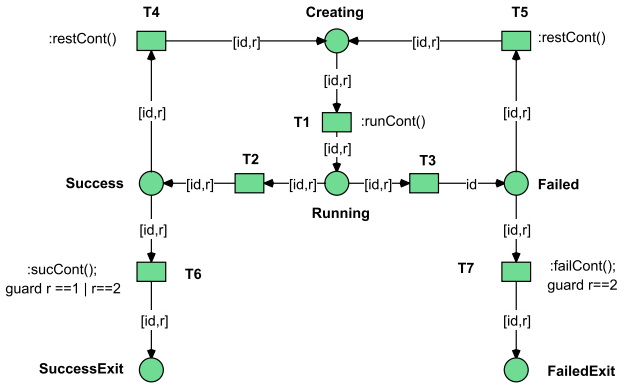

Figure 2: Model of the lifecycle of a Container

ken Nets. The tokens inside our token net are containers and the tokens inside our System Net represent Pods, as illustrated in Figure 1. The details of how to create the instances of the Token nets are hidden to improve legibility. In addition, we have supposed that the scheduler assigns a pod to a single node arbitrarily, as long as the machine has enough resources available. If there are not enough resources in the cluster, the pod waits in **Pending Scheduling** place. This behaviour could be refined by introducing more sophisticated policies and a reject place to pods. The place **Machines** represents the resources managed by the scheduler. For each machine, there is a tuple token with the identification of the node, the amount of available RAM and the number of available cores. The resources assigned to a pod are only released when the pod restart policy is "never" or "onFailure".

Once the pod has been assigned to a machine, Kubernetes starts creating the containers (synchronised in the model by the inscription *runCont* with the Token Net in Figure 2) and the pod waits in **Pending** place. When all containers have been created, the pod state changes to **Running**. While in **Running** state, the pod waits for its containers to terminate. If a container fails, the pod goes to **RunningFailed** place where it waits for the termination of all containers and based on the restart policy, the containers might be restarted. If there are no failures, the pod will be in **Running** place or eventually will reach **Success** place when all containers have finished.

Figure 2 illustrates the behaviour of a container (without places and transitions needed to create the initial marking, as before). Tokens in the net are the identifiers for each container. For simplicity, we have included in the net the restart policy. A created pod enters the **Running** place, and may reach the **Success** or **Failure** place. The firing of the corresponding transitions (**T1**, **T2** and **T3**) is synchronised with the System Net. According to the restart policy, the containers might return to **Running** place or they might finish in **SuccessExit** or in **FailedExit** places. We include several timed transitions, as summarised in Table 1. By default, the firing of **T2** and **T3** is arbitrary and non-deterministic; however, with Renew, it is possible to simulate any probability distribution for **T2** and **T3** transitions in order to simulate a failure. Additionally, it is possible to assign different random distributions for the timed transitions. In the next sections we have made different experiments to obtain the real value of these metrics. **T2** and **T3** transitions are application dependant (they represent the termination time and the time to the next failure, respectively). The termination time (**T6** and **T7**) and the restarting time for a container (**T4** and **T5**) does not depend on the success of the container, so both transitions are modelled with the same distribution. Transitions in Token Net are synchronised with the System Net. For instance, when a container is terminated, the corresponding pod in the System Net is moved in the high level Net.

Table 1: Timed transitions in the model

Transition	Variable
T1	Time to create a container
T2	Execution time of a container
T3	Time until next failure in a container
T4, T5	Time to restart a container.
T6, T7	Graceful Termination of a container

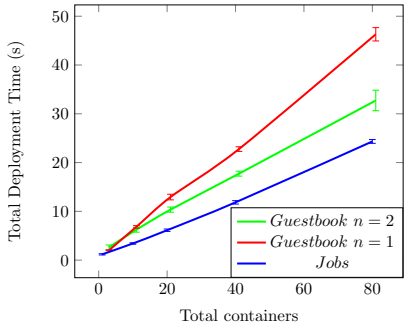

Figure 3: Deployment time vs. Number of Pods with a 95% confidence interval.

3.2 Creation time

We deploy Kubernetes on two physical machines, each with 32GB of RAM and 12 Intel Xeon E5-2620 (2.00GHz) cores. In Figure3, we show the mean deployment time of the Guestbook application with different configurations with one and two machines (n=1, n=2). Each machine has preloaded docker images. In Guestbook experiments, each pod has exactly one container. The "jobs" line in the figure is the same experiment with all containers in a single pod. From the result, we can observe the overhead introduced by creating pods over the docker containers. Figure 4 depicts the mean time needed to create a container per machine in each scenario, illustrating $\sim$ 0,6s deployment time for a single machine and 1s for two machine. A possible explanation of this behaviour could be the overhead introduced to synchronize the deployment in different machines. With this experiment, we can estimate the value of **T1** depending of the scenario.

3.3 Termination Time

A Pod is expected to be terminated at some time. If it is a service, and consequently it has to be running all the time, the termination might due to a failure and the pod has to be restarted. This philosophy also applied to containers, as we have discussed previously in the Container Net model (transitions **T4**, **T5**, **T6**, **T7**). We can consider that **T3** depends on the application, and it represents the failure rate (or the

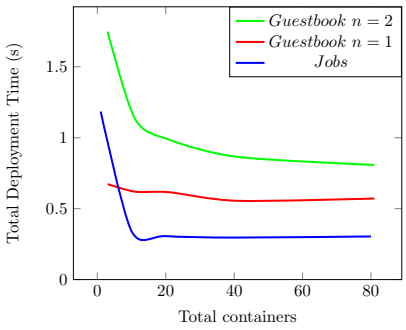

Figure 4: Mean Deployment time per container per machine vs. number of containers.

Table 2: T5 and T6 experimental results

C	**T5** per Container	**T6** Graceful termination	**T6** per Container
1	0.01	30	0
10	0.11	30.99	0.10
40	0.15	34.69	0.11
60	0.16	37.04	0.11

time between failures). When a pod terminates, Kubernetes waits a grace period (which by default is 30 seconds) until it kills any associated container and data structures. To associate monitored metrics with transitions, we perform the following experiments:

Transitions **T4, T5**: relate to container re-start time. We have deployed pods with a variable number of containers to measure this time. Results are shown in column "**T5** per Container" in Table 2. The mean time to terminate a container is independent of the number of containers in a pod. Additionally, the highest experimental measure is ~ 0.3s and the 80 % of the sampled times are <0.22s. The behaviour of **T4** is the same that **T5**

Transitions **T6, T7**: model the normal behaviour of Kubernetes. On successful completion, Kubernetes waits for the grace period and deletes all data structures associated with a container. We have measured these variables in columns "**T6** Graceful termination" and "T6 per container" in Table 2. For these experiments we have set the grace period to 30s (the default). We can observe that the overhead (Column "**T6** per container") is independent of number of containers. Transition **T7** has a similar behaviour.

3.4 Micro-benchmarks

To measure overheads for applications deployed in Kubernetes, we consider the following scenarios: (i) one pod is deployed and all containers are inside that pod; (ii) several pods are deployed and there is exactly one container inside the pod. The total number of containers deployed is given by C. All pods are deployed on the same machine. Each experiment has been repeated twenty times and we present the mean (μ_i) and standard deviation (σ_i) for scenario i. For comparison, we consider the following test: $H_0 : \mu_1 - \mu_2 = 0$ and $H_1 : \mu_1 - \mu_2 \neq 0$.

Table 3: Pov-ray experiment. Execution time (T2 in the model) for Scenario 1 and Scenario 2 & hypothesis testing.

	Scenario 1		Scenario 2		
C	μ_1	σ_1	μ_2	σ_2	$\mu_1 - \mu_2 = 0$?
1	123.47	0.43	123.38	0.39	Yes
4	473.65	0.96	475.15	0.62	No
8	946.90	0.72	946.63	0.69	Yes
12	1417.76	1.67	1420.40	1.35	No
20	2370.21	1.16	2374.36	3.89	Yes

Table 4: Execution time (in seconds) of bzip benchmark for Scenario 1 and Scenario 2 & hypothesis testing

	Scenario 1		Scenario 2		
C	μ_1	σ_1	μ_2	σ_2	$\mu_1 - \mu_2 = 0$?
1	16.05	0.16	15.98	0.24	Yes
4	16.94	0.15	16.896	0.16	Yes
8	19.20	1.50	19.86	0.63	Yes
12	20.63	1.52	20.35	1.17	Yes
20	36.35	2.69	35.26	0.99	Yes
40	67.85	2.59	69.21	1.35	Yes

CPU intensive application: we use the multi-threaded pov-ray 3.7 benchmark to measure overhead of pods for CPU intensive use. Kubernetes permits the Docker-based CPU reservation for containers. If there are no contingency cases, a container uses all available CPU cores. For multiple CPUs, sharing is proportion to reservations. The results of the experiments are shown in Table 3. As expected, the execution time is linear to the number of parallel containers/pods executing pov-ray benchmark. With a single container, all the CPU is used by the application. As the machine has 12 cores, the performance of twelve containers should be similar to the performance of pov-ray executed on a Docker container with multi-threading disabled. With this metric as reference value, we can calculate the overhead introduced by Kubernetes in CPU usage (about 14%). With these results, it seems that, for CPU intensive applications, it is a better solution to group all containers in a same pod to reduce the overhead.

I/O intensive: we use BZip as a representative benchmark of an I/O application. Table 4 shows the results, measuring execution time of n containers for compressing the Linux kernel (about 98MB) with bzip. For all set of experiments, we have accepted H_0. With these results, we can conclude that the performance of I/O applications is not affected for the deployment scenario. Although the access to the file system is shared for all containers in a pod, the overhead of that shared resource is negligible.

Network application: containers inside a pod share a network connection. To determine the impact of this for each container, we deploy an `iperf` server in a pod and several clients with the previous scenarios configuration. All tests are run for 30s for TCP-based traffic. In Scenario 1, all containers are inside the pod; in Scenario 2, all clients are inside a pod, and all of them are scheduled to the same machine

Table 5: Network bandwidth with C iperf Clients in Scenario 1. Iperf server & Iperf client are on the same machine.

C	$\mu_1(GB)$	σ_1	$\sum BW_i/C(GB)$
1	1.88	0.06	1.88
4	8.61	0.21	2.15
8	15.53	0.12	1.94
12	14.99	0.21	1.25

Table 6: Network bandwidth with C iperf Clients in Scenario 2. Iperf server & Iperf client are in the same machine. $H_0 : \mu_1 - \mu_2 = 0$?

C	$\mu_2(GB)$	σ_2	$\sum BW_i/C(GB)$	H_0?
1	1.90	0.04	1.90	Yes
4	8.82	0.05	2.20	Yes
8	16.26	0.20	2.03	No
12	16.42	0.38	1.37	No

to avoid the impact of the network. The hypothesis test is stated from the summary of the bandwidth of all containers in Scenario 1 and of all pods in Scenario 2. Results are shown in Table 5 and Table 6. From the results, we can observer that for application with more than eight containers, the bandwidth per container is better when we deploy each container in an isolated pod. This suggests that deploying several pods with a few coupled containers is better than a single pod with a large number of containers.

4. CONCLUSION

A performance model for Kubernetes based deployment is outlined. With emerging interest in applications (such as stream processing) which need to launch and terminate instances on a per second basis, the overhead associated with VMs remains a limitation. We use a benchmark-based approach to better characterise behaviour of a Kubernetes system (using Docker containers). We propose a Reference net-based model for Pod & container lifecycle in Kubernetes. Such a model can be used as a basis to support: (i) capacity planning and resource management; (ii) application design, specifically how an application may be structured in terms of pods and containers. Our future work involves undertaking further analysis on the model to study its properties and infer potential behaviours to support application scaling.

Acknowledgments

This work was supported in part by: The Industry and Innovation department of the Aragonese Government and European Social Funds (COSMOS group, ref. T93) and the Spanish Ministry of Economy (*Programa de I+D+i Estatal de Investigación, Desarrollo e innovación Orientada a los Retos de la Sociedad* âĂŞTIN2013-40809-R). V. Medel was the recipient of a fellowship from the Spanish Ministry of Economy and from the Fundación Ibercaja-CAI.

5. REFERENCES

[1] S. Brunner, M. Blochlinger, G. Toffetti, J. Spillner, and T. M. Bohnert, "Experimental evaluation of the cloud-native application design," *IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pp. 488–493, 2015.

[2] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[3] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273025

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, March 2015, pp. 171–172.

[5] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *14th IEEE International Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, 2015, pp. 27–34.

[6] H. Khazaei, J. Misic, and V. Misic, "Performance analysis of cloud computing centers using m/g/m/m+r queuing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 5, pp. 936–943, 2012.

[7] R. Tolosana-Calasanz, J. Á. Bañares, and J. M. Colom, "Towards Petri net-based economical analysis for streaming applications executed over cloud infrastructures," in *Economics of Grids, Clouds, Systems, and Services - 11th International Conference, GECON'14, Cardiff, UK, September 16-18, 2014.*, ser. LNCS, vol. 8914, 2014, pp. 189–205.

[8] A. Merino, R. Tolosana-Calasanz, J. Á. Bañares, and J. M. Colom, "A specification language for performance and economical analysis of short term data intensive energy management services," in *Economics of Grids, Clouds, Systems, and Services - 12th International Conference, GECON 2015, Cluj-Napoca, Romania, September 15-17, 2015*, ser. LNCS, vol. 9512, 2015, pp. 147–163.

[9] J. Hwang, S. Zeng, F. y Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, 2013, pp. 269–276.

[10] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing," in *Information, Electronic and Electrical Engineering, 2015 IEEE 3rd Workshop on Advances in.* IEEE, 2015, pp. 1–8.

[11] R. Valk, "Object petri nets: Using the nets-within-nets paradigm, advanced course on petri nets 2003 (j. desel, w. reisig, g. rozenberg, eds.), 3098," 2003.

[12] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, and R. Valk, "An extensible editor and simulation engine for petri nets: Renew," in *International Conference on Application and Theory of Petri Nets*. Springer, 2004, pp. 484–493.